平成 30 年度

筑波大学情報学群情報科学類

卒業研究論文

題目
グループ化機能を持つ統合言語クエリの設計と効率的実装のためのプログラム変換

主専攻　　知能情報メディア主専攻

著者　　大倉 瑠維

指導教員　亀山幸義, 海野広志

要　旨

本研究の目的は、SQL のグループ化に直接対応する統合言語クエリを設計することであり、設計した構文の意味論や書き換え規則を追加することである。

SQL におけるグループ化は、グループ化前後でデータの型を変化させてしまうだけでなく、サブクエリのようなネストされた構造を用いることでしか表現できないクエリが存在するため、統合言語クエリにおけるグループ化の拡張は困難とされてきた。本研究では、従来の **for** 項にグループ化機能を付与した **gfor** 項といった構文を新たに提案し、束縛変数の追加や条件式の組み込み等により、一つの **gfor** 項に対して単一のグループ化機能つき SQL クエリに対応するように設計した。グループ化の拡張に伴い新たに追加した書き換え規則による書き換えについては、複数のテーブルに対する処理の仕方や、グループ化キーを増やす処理等、従来とは少し異なる視点で書き換え規則を追加することによって、書き換えを可能にした。また、集約関数について、代数的に使えるもののみに制限することで、書き換えによる集約関数の正規化を可能にした。

目次

図目次

表目次

第 1 章

はじめに

1.1 背景

データベースを活用するシステムでは、データベース問い合わせ言語とプログラム言語の 2 つの言語を扱う必要がある。しかし、2 つの言語を扱うことは、コードの複雑化やセキュリティ上の問題により様々な負担をプログラマに強いることになるため、20 年以上前から問題とされてきた。データベース問い合わせ言語である SQL では、抽象や合成、入れ子のデータ構造等を直接表現できないが、静的型付き言語のようなプログラム言語ではこれらの表現を可能とし、データベース問い合わせ言語とプログラム言語を 1 つの言語として統合することでプログラマの負担を減らした効率の良いシステムが開発できると期待されている。1 つに統合された言語を統合言語クエリと呼び、代表的なものとして Microsoft LINQ 等が挙げられる [9]。LINQ 等の従来の統合言語クエリでは、様々なデータ構造を持つシステムに対して効率よくアクセスできるようになったが、複数のクエリ項の合成を許すことでクエリ雪崩 [1] という問題を発生させた。クエリ雪崩とは、コード変換によって多数のクエリを生成してしまうという問題であり、生成したクエリの数だけデータベースへアクセスするという点から実行効率を大幅に下げてしまう。

この問題に対し、Cheney らはクエリ項を直接実行せずに、生成したコードを書き換え規則に基づいて書き換えを行うことで単一の SQL クエリを生成し、クエリ雪崩を防ぐことに成功した [2]。一方で、書き換え規則において型付けが保存されるかは手動で証明されている。そのため、書き換え規則の追加や構文の拡張をする際には手動で証明しなければいけないため、拡張は困難である。

そこで、Suzuki らは、終タグレス法 [3] に基づき、型安全性をホスト言語の型付けに帰着させることで、書き換え規則の型付け保存性を自動的に検査されるようにした [4]。これにより、まだ拡張されていない構文に対して拡張可能であることを提唱した。しかし、SQL でのグループ化において、OCaml や Haskell のような関数型言語で表現しようとするとグループ化前後でデータの型が保存されない [5] だけでなく、既存の構文の拡張では意味論や書き換え規則の等価性を表すことができないため、統合言語クエリのグループ化に対する拡張は困難とされてきた。

1.2 本研究の目的と構成

本研究では、SQL でのグループ化に対して新たな統合言語クエリの構文を設計し、その構文に対する型付け規則、意味論、書き換え規則等を提案する。変換の対象となる言語は Suzuki らの提案した QueΛ とし、先行研究と

同様にクエリの抽象や合成が可能で、クエリ雪崩を起こさない。本研究の貢献や従来の研究と異なる点としては以下の点が挙げられる。

- グループ化機能付き SQL の構文に直接対応するようなグループ化の構文を提案し、追加した書き換え規則を用いて書き換えを行うことで、単一の SQL クエリを生成することを可能にした。
- 複数のテーブルに対する解釈を変更し、従来の構文では書けなかった書き換え規則の記述を可能にした。
- ネストされた構造を一部許すことで、SQL においてサブクエリを使うことでしか表すことのできない構文に直接対応できるようにした。

本論文の構成は次の通りである。まず、第 2 章で前提知識として、コードの正規化とグループ化について概説する。第 3 章においては、本研究での対象言語 QueΛ の型付け規則や意味論、SQL 変換について記述する。第 4 章では、本研究において新たに提案するグループ化の構文 **gfor** をはじめ、グループ化に関する構文全般の型付け規則や意味論や正規形等について記述する。また、同時にグループ化において不可欠となる集約関数の働きについても記述する。第 5 章では、本研究のまとめと今後の課題について述べる。

第 2 章

前提知識

本章では、前提知識として、クエリの正規化の必要性とグループ化の難点について例題を用いて概説する。

2.1　正規化

統合言語クエリにおけるクエリ雪崩の問題を解決するために、Cheney ら [2] はクエリをコードとして生成した後、正規化規則によってコードを書き換えることで単一の SQL クエリを生成することに成功した。この書き換えを正規化と呼び、正規化の必要性を図 2.1 のテーブルを用いて述べる。

en_test テーブル

date	sid	name	results
2018-10-01	1	Sato	30
2018-10-01	2	Suzuki	50
2018-10-02	1	Sato	40
2018-10-02	2	Suzuki	45
2018-10-02	3	Takahashi	60
2018-10-03	3	Takahashi	85

class テーブル

sid	class
1	A
2	B
3	A
4	B
5	C

図 2.1　データテーブル

en_test テーブルは全生徒の英語の小テスト情報として、テスト受験日 (date), 生徒 ID(sid), 生徒名 (name), テスト結果 (results) を持つ。また、class テーブルは生徒のクラス情報として、生徒 ID(sid), クラス (class) を持つ。

データベース上のテーブルのデータ構造は、$Int, String$ の基本的な型を持つレコードのバッグとして表現できる。例えば、図 2.1 の en_test テーブルの型を $EnTest$ とすると、型 $EnTest$ は以下のように表せる。なお、SQL には $Date$ 型という日付の型を持つデータ型が存在するが、ここでは簡単のために日付は $String$ 型で表すこととする。

$$EnTest = \langle \mathsf{date} : String, \mathsf{sid} : Int, \mathsf{name} : String, \mathsf{results} : Int \rangle$$

この型に基づき、図 2.1 のデータ構造は、図 2.2 のように記述できる。

$edata$: Bag $EnTest$ =
[⟨ date = “2018-10-01”, sid = 1, name = “Sato”, results = 30 ⟩;
⟨ date = “2018-10-01”, sid = 2, name = “Suzuki”, results = 50 ⟩;
⟨ date = “2018-10-02”, sid = 1, name = “Sato”, results = 40 ⟩;
⟨ date = “2018-10-02”, sid = 2, name = “Suzuki”, results = 45 ⟩;
⟨ date = “2018-10-02”, sid = 3, name = “Takahashi”, results = 60 ⟩;
⟨ date = “2018-10-03”, sid = 3, name = “Takahashi”, results = 85 ⟩]

$cdata$: Bag $Class$ =
[⟨ sid = 1, class = “A” ⟩;
⟨ sid = 2, class = “B” ⟩;
⟨ sid = 3, class = “A” ⟩;
⟨ sid = 4, class = “B” ⟩;
⟨ sid = 5, class = “C” ⟩]

図 2.2　図 2.1 のデータ構造

ここで、2 つのクエリを合成することで起きるクエリ雪崩と、その解決方法であるコードの正規化を説明するために、2 つのクエリ (Q_1, Q_2) を用意する。クエリ Q_1 は、指定したクラスに該当する生徒を返す操作をするクエリであり、クエリ Q_2 は、指定された生徒 ID に該当し、かつ指定した日付の生徒の英語の点数を返す操作をするクエリである。内包表記を用いて記述したクエリとそれに対応する SQL クエリを図 2.3 に示す。

(クエリ Q_1)
$Q_1 = \lambda class.$
　　for $(c \leftarrow$ **table**(“class”))
　　where $(c$.class $= class)$
　　yield c

```
SELECT c.*
FROM class AS c
WHERE c.class = class
```

(クエリ Q_2)
$Q_2 = \lambda c.\ \lambda date.$
　　for $(e \leftarrow$ **table**(“en_test”))
　　where $(e$.sid $= c$.sid $\wedge\ e$.date $= date)$
　　yield ⟨ date $= e$.date,
　　　　name $= e$.name,
　　　　results $= e$.results,
　　　　class $= c$.class ⟩

```
SELECT e.date AS date,
       e.name AS name,
       e.results AS results,
       c.class AS class
FROM en_test AS e
WHERE e.sid = c.sid AND
       e.date = date
```

図 2.3　クエリ Q_1, Q_2 とそれぞれに対応する SQL クエリ

内包表記 **for** $(x \leftarrow M)\ N$ は、データベース上のテーブルに対応するバッグ M から要素を一つずつ取り出し、変数 x に束縛した環境の下で項 N を評価する。**table** 構成子はテーブル名を受け取り、そのテーブルをレコードのバッグとして返す働きをする。**where** $L\ M$ は、L が **true** であれば、M を評価した結果を返し、それ以外であれば空のバッグを返す。**yield** M は、M を評価した結果を単一の要素とするバッグを返す。

ここで、2 つのクエリを合成する関数 $compose$ を用意する。クエリ Q_1 を実行して得られた N 件のデータに対して、クエリ Q_2 を N 回実行するような合成関数を仮に用意すると、N+1 件のクエリが発行されてしまうため、クエリ雪崩を起こす。このクエリ雪崩の現象を回避しつつ、クエリを合成する関数を以下に示す。

$$
\begin{array}{rcl}
compose & = & \lambda q.\ \lambda r.\ \lambda x.\ \lambda z. \\
 & & \quad \textsf{for}\ (y\ \leftarrow\ q\ x)\ r\ y\ z
\end{array}
$$

関数 $compose$ を用いてクエリ Q_1, Q_2 を合成したクエリを Q_3 とする。しかし、Q_3 はそのままでは直接 SQL に変換することができない。そのため、正規化を行うことで単一の SQL クエリに変換する。クエリ Q_3 と正規化後の Q_3 を図 2.4 に示す。

$$
\begin{array}{rcl}
Q_3 & = & \lambda x.\ \lambda z.\ compose\ Q_1\ Q_2\ x\ z \\
 & = & \lambda x.\ \lambda z. \\
 & & \quad \textsf{for}\ (y\ \leftarrow\ (\lambda class. \\
 & & \qquad \textsf{for}\ (c \leftarrow \textsf{table}(\text{“class”})) \\
 & & \qquad \textsf{where}\ (c.\textsf{class}\ =\ class) \\
 & & \qquad \textsf{yield}\ c)\ x) \\
 & & \quad (\lambda c.\ \lambda date. \\
 & & \quad \textsf{for}\ (e \leftarrow \textsf{table}(\text{“en\_test”})) \\
 & & \quad \textsf{where}\ (e.\textsf{sid}\ =\ c.\textsf{sid}\ \wedge\ e.\textsf{date}\ =\ date) \\
 & & \quad \textsf{yield}\ \langle\ \textsf{date}\ =\ e.\textsf{date}, \\
 & & \qquad \textsf{name}\ =\ e.\textsf{name}, \\
 & & \qquad \textsf{results}\ =\ e.\textsf{results}, \\
 & & \qquad \textsf{class}\ =\ c.\textsf{class}\ \rangle)\ y\ z
\end{array}
$$

↓ 正規化

$$
\begin{array}{rcl}
Q_3 & = & \textsf{for}\ (c \leftarrow \textsf{table}(\text{“class”})) \\
 & & \textsf{for}\ (e \leftarrow \textsf{table}(\text{“en\_test”})) \\
 & & \textsf{where}\ (c.\textsf{class}\ =\ \text{“}A\text{”}\ \wedge\ e.\textsf{sid}\ =\ c.\textsf{sid}\ \wedge\ e.\textsf{date}\ =\ \text{“2018-10-02”}) \\
 & & \textsf{yield}\ \langle\ \textsf{date}\ =\ e.\textsf{date}, \\
 & & \qquad \textsf{name}\ =\ e.\textsf{name}, \\
 & & \qquad \textsf{results}\ =\ e.\textsf{results}, \\
 & & \qquad \textsf{class}\ =\ c.\textsf{class}\ \rangle
\end{array}
$$

↓ SQL 変換

```
SELECT e.date AS date, e.name AS name,
       e.results AS results, c.class AS class
FROM en_test AS e, class AS c
WHERE c.class = "A" AND e.sid = c.sid AND e.date = "2018-10-02"
```

図 2.4　クエリ Q_3 の正規化と SQL 変換

このように、正規化を行うことで、クエリ雪崩を起こすことなく、単一の SQL クエリを生成することができる。また、正規化を前提とすることで、中間のデータ構造をネストすることができ、より見通しの良いクエリを記述することができる。

2.2 グループ化

SQL におけるグループ化とは、データベース上のテーブルにおいて同列内で同じ値を持つデータをまとめることを言い、**GROUP BY** 句を用いる。前節の図 2.1 に対して、date でグループ化し、その日の英語の最高点を返す SQL クエリは以下のように書くことができる。

```
SELECT e.date AS date, MAX(e.results) AS max
FROM en_test AS e
GROUP BY e.date
```

一方で、上記のようなグループ化をする SQL クエリに直接対応する統合言語クエリは我々の知る限り存在しない。その理由として、以下の点が挙げられる。

1. 既存言語での構文の拡張が困難
2. 正規化できないクエリの組み合わせが存在

1 について、既存の統合言語クエリの構文において単純にグループ化機能のみを拡張すると、複数のテーブルに対してグループ化する場所次第で意味が異なってしまう。そのため、意味論の記述が困難となり、既存の構文にグループ化機能のみをそのまま拡張することは困難である。

2 について、例えばグループ化したクエリをさらにグループ化するという操作に対して、SQL では必ずサブクエリを用いて表現しなければならない。サブクエリとは、クエリ内部に入れ子になっているクエリのことであり、SQL でサブクエリを使用するときにはサブクエリによって作成された仮のテーブルを用いて、外側のクエリを評価する。ただ、サブクエリを用いることは実行効率を遅らせるとされているため、Cheney らの先行研究ではサブクエリ付きのクエリは、正規化によってフラットな構造のクエリに変換される。本研究ではグループ化機能によって表されるクエリのすべてを表現するため、グループ化の操作のみサブクエリを許す必要がある。また、項 **for** に単純にグループ化機能を付与した構文を **group** とすると、**for** と **group** の組み合わせに対して既存の書き換え規則の書き換えではどれも正規化できないため、異なるアプローチで書き換え規則を追加しなければならない。これらの観点からグループ化は困難とされ、本研究ではこれらの問題に対して、新たな提案方式を提唱し、グループ化機能を持つ統合言語クエリを設計する。

第 3 章

統合言語クエリ Queλ

本研究では、Suzuki らが提案した Queλ を対象言語とした変換を定義する。Queλ は Cooper の言語 [6] からエフェクトを除いたものであり、Cheney らの T-LINQ からコード化の機能を除いたものに似ている。Queλ の構文を図 3.1 に示す。

Syntax

Base types	O	::=	$Int \mid Bool \mid String$	
Types	A, B	::=	O	
		$\mid$	$A \to B$	(function type)
		$\mid$	$\mathsf{Bag}\ A$	(bag)
		$\mid$	$\langle \overrightarrow{l : A} \rangle$	(record)
Type environment	Γ	::=	ϕ	(empty environment)
		$\mid$	$\Gamma, x : A$	(term variable binding)
Primitives	$\oplus$			
Table	s, t			
Field names	l			
Terms	L, M, N	::=	$\lambda x.\ M$	(λ-abstraction)
		$\mid$	$M\ N$	(application)
		$\mid$	$\oplus(\overrightarrow{M})$	(primitive)
		$\mid$	x	(variable)
		$\mid$	c	(constant)
		$\mid$	$\mathbf{for}\ (x \leftarrow M)\ N$	(comprehension)
		$\mid$	$\mathbf{where}\ L\ M$	(test)
		$\mid$	$\mathbf{yield}\ M$	(singleton)
		$\mid$	$[\,]$	(empty)
		$\mid$	$M \uplus N$	(bag union)
		$\mid$	$\mathbf{exists}\ M$	(not empty)
		$\mid$	$\mathbf{table}(t)$	(table reference)
		$\mid$	$\langle \overrightarrow{l = M} \rangle$	(record constructor)
		$\mid$	$L.l$	(projection)

図 3.1　Queλ の構文

型は、$Int, Bool, String$ からなる基本型 O と、関数型やバッグ、レコードからなる A, B がある。型環境は Γ で表し、拡張する場合はカンマの右側に新たな変数を加える。$\oplus$ はプリミティブな演算子を表し、テーブルやレコードラベルとして t、l がある。項はバッグ内包表記 **for** $(x \leftarrow M)\ N$、条件分岐 **where** $L\ M$、単一要素のバッグを返す **yield** M だけでなく、λ 抽象や関数適用がある。また、バッグが空かどうかを判定するものは **exists** M で表し、データベースのテーブルを参照するものは **table**(t) として表す。レコードは $\langle \overrightarrow{l = M} \rangle$、射影は $L.l$ と表す。

3.1 QueΛ の型付け規則

前節における QueΛ の構文の型付け規則を図 3.2 に表す。

$\boxed{\Gamma \vdash M : A}$

CONST
$$\frac{\Sigma(c) = O}{\Gamma \vdash c : O}$$

OP
$$\frac{\Sigma(\oplus) = O_1 \times ... \times O_n \to O \quad \Gamma \vdash M_i : O_i \ (\text{for each } 1 \leq i \leq n)}{\Gamma \vdash \oplus(\overrightarrow{M}) : O}$$

VAR
$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

ABS
$$\frac{\Gamma, x : A \vdash N : B}{\Gamma \vdash \lambda x.N : A \to B}$$

APP
$$\frac{\Gamma \vdash L : A \to B \quad \Gamma \vdash M : A}{\Gamma \vdash L\ M : B}$$

RECORD
$$\frac{\Gamma \vdash M_i : A_i (\text{for each } 1 \leq i \leq n)}{\Gamma \vdash \langle \overrightarrow{l = M} \rangle : \langle \overrightarrow{l : A} \rangle}$$

PROJECT
$$\frac{\Gamma \vdash L : \langle \overrightarrow{l : A} \rangle}{\Gamma \vdash L.l_i : A}$$

SINGLETON
$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \textbf{yield}\ M : \mathsf{Bag}\ A}$$

EMPTY
$$\frac{}{\Gamma \vdash [\,] : \mathsf{Bag}\ A}$$

UNIONALL
$$\frac{\Gamma \vdash M : \mathsf{Bag}\ A \quad \Gamma \vdash N : \mathsf{Bag}\ A}{\Gamma \vdash M \uplus N : \mathsf{Bag}\ A}$$

TABLE
$$\frac{\Sigma(t) = \mathsf{Bag}\ \langle \overrightarrow{l : O} \rangle}{\Gamma \vdash \textbf{table}(t) : \mathsf{Bag}\ \langle \overrightarrow{l : O} \rangle}$$

FOR
$$\frac{\Gamma \vdash M : \mathsf{Bag}\ A \quad \Gamma, x : A \vdash N : \mathsf{Bag}\ B}{\Gamma \vdash \textbf{for}(x \leftarrow M)\ N : \mathsf{Bag}\ B}$$

WHERE
$$\frac{\Gamma \vdash L : Bool \quad \Gamma \vdash M : \mathsf{Bag}\ A}{\Gamma \vdash \textbf{where}\ L\ M : \mathsf{Bag}\ A}$$

EXISTS
$$\frac{\Gamma \vdash M : \mathsf{Bag}\ A}{\Gamma \vdash \textbf{exists}\ M : Bool}$$

図 3.2　QueΛ の型付け規則

CONST 規則は定数項 c の型が基本型 O であるならば、その定数項は型 O を持つという意味である。OP 規則はプリミティブ演算子の型がそれぞれ異なる型 $O_1 \times ... \times O_n$ からある型 O への関数型であり、かつ型環境 Γ の下で n 個の項がそれぞれプリミティブ演算子の引数の型に対応する型を持つならば、$\oplus(\overrightarrow{M})$ は型 O を持つという意味である。他にも、FOR 規則は M が型 Bag A を持ち、型環境 Γ と型 A を持つ変数 x の下で N が型 Bag B を持つならば、**for** $(x \leftarrow M)\ N$ は型 Bag B を持つという意味である。

3.2 QueΛ の意味論

次に、QueΛ の値、評価文脈、操作的意味論を図 3.3 に定義する。

Value and Evaluation Contexts

$$
\begin{array}{lrcl}
\text{Value} & V, W & ::= & c \mid \lambda x.M \mid \langle \overrightarrow{l = V} \rangle \mid [\, V \,] \\
\text{Evaluation Contexts} & \mathcal{E} & ::= & [\,] \mid \oplus(\overrightarrow{V}, \mathcal{E}, \overrightarrow{M}) \mid \mathcal{E}\ M \mid V\ \mathcal{E} \\
 & & \mid & \langle \overrightarrow{l = V}, l' = \mathcal{E}, \overrightarrow{l'' = M} \rangle \mid \mathcal{E}.l \mid \textsf{yield}\ \mathcal{E} \\
 & & \mid & \mathcal{E} \uplus M \mid V \uplus \mathcal{E} \mid \textsf{for}\ (x \leftarrow \mathcal{E})\ N \mid \textsf{exists}\ \mathcal{E} \\
 & & \mid & \textsf{where}\ \mathcal{E}\ M
\end{array}
$$

Operational Semantics

$$
\begin{array}{rcll}
\oplus(\overrightarrow{V}) & \longrightarrow & \sigma(\oplus, \overrightarrow{V}) & \text{(E-OP)} \\
(\lambda x.N)\ V & \longrightarrow & N[x := V] & \text{(E-ABS-}\beta\text{)} \\
\langle \overrightarrow{l = V} \rangle.l_i & \longrightarrow & V_i & \text{(E-RECORD-}\beta\text{)} \\
\textsf{where true}\ M & \longrightarrow & M & \text{(E-WHERETRUE)} \\
\textsf{where false}\ M & \longrightarrow & [\,] & \text{(E-WHEREFALSE)} \\
\textsf{for}\ (x \leftarrow \textsf{yield}\ V)\ M & \rightsquigarrow & M[x := V] & \text{(E-FORYIELD)} \\
\textsf{for}\ (x \leftarrow [\,])\ N & \longrightarrow & [\,] & \text{(E-FOREMPTY}_1\text{)} \\
\textsf{for}\ (x \leftarrow L \uplus M)\ N & \longrightarrow & & \\
 & & \textsf{for}\ (x \leftarrow L)\ N \uplus \textsf{for}\ (x \leftarrow M)\ N & \text{(E-FORUNION}_1\text{)} \\
\textsf{exists}\ [\,] & \longrightarrow & \textsf{false} & \text{(E-FOREXISTFALSE)} \\
\textsf{exists}\ [\overrightarrow{V}] & \longrightarrow & \textsf{true} \quad (|\overrightarrow{V}| > 0) & \text{(E-EXISTTRUE)}
\end{array}
$$

$$
\frac{M \longrightarrow N}{\mathcal{E}[M] \longrightarrow \mathcal{E}[N]}\ \text{(E-CONTEXT)}
$$

図 3.3　QueΛ の値、評価文脈、操作的意味論

定数、λ 抽象、レコード、レコードのバッグが値となる。[V] は、**yield** $V_1 \uplus \ldots \uplus$ **yield** $V_n \uplus [\,]$ の略記とし、左側の項から評価される。評価文脈では、図 3.1 で定義した QueΛ の構文に対してプリミティブ演算子、関数適用、バッグの結合が左側から評価される。操作的意味論では、σ は各々のプリミティブ演算子に対する解釈であり、σ によって意味論をパラメータ化している。E-FORUNION$_1$ は、レコードのバッグ同士を $\uplus$ で結合してから評価を進める操作に対し、SQL クエリに直接変換するために、それぞれのレコードのバッグの評価をしてから $\uplus$ によってレコードのバッグを結合する操作に書き換えている。これは正規化規則としても定義される。

3.3 SQL 変換

次に、QueΛ によって書かれたクエリを SQL に変換する対象となる SQL サブ言語と変換規則を図 3.4、図 3.5 に示す。SQL クエリでは、基本的にそれぞれの問い合わせ式を記述する **SELECT** 句、テーブルとその束縛を記述する **FROM** 句、条件式を記述する **WHERE** 句からなる。QueΛ において **for** $(x \leftarrow M)\ N$ を連続して書いたもの

は複数のテーブルに対する操作であるため、SQL クエリでは **FROM** 句にて並べて記述され、そこで複数のテーブルの内部結合が行われる。また、QueΛ におけるプリミティブ演算は SQL クエリでは $\oplus_{sql}$ と書くこととし、変数 x のレコード全てに対応するものは、SQL において要素すべてという意味である $x.*$ と書く。図 3.5 での $where$ の条件式は一部の項に対して事前に SQL 変換を行った結果を表しており、その結果を用いてクエリを変換する。

$$
\begin{array}{rcl}
P, Q & := & P \ \textsf{UNION ALL} \ Q \mid S \\
S & := & \textsf{SELECT} \ \overrightarrow{s} \ \textsf{FROM} \ t \ \textsf{AS} \ x \ \textsf{WHERE} \ e \\
s & := & e \ \textsf{AS} \ l \mid x.* \\
e & := & c \mid x.l \mid e \wedge e' \mid \neg e \mid \textsf{EXISTS}(Q) \mid \oplus_{sql} (\overrightarrow{e})
\end{array}
$$

図 3.4　SQL サブ言語

$$
\begin{array}{rcl}
S[\![U_1 \uplus U_2]\!] & = & S[\![U_1]\!] \ \textsf{UNION ALL} \ S[\![U_2]\!] \\
S[\![[\,]]\!] & = & \textsf{SELECT} \ \overrightarrow{null \ \textsf{AS} \ l} \ \textsf{FROM} \ \phi \ \textsf{WHERE FALSE} \\
S[\![\textsf{for} \ (x \leftarrow \textsf{table}(s)) \ F]\!] & = & \textsf{SELECT} \ \overrightarrow{e \ \textsf{AS} \ l} \ \textsf{FROM} \ s \ \textsf{AS} \ x, \overrightarrow{t \ \textsf{AS} \ y} \ \textsf{WHERE} \ B \\
 & & where \ S[\![F]\!] \ = \ (\textsf{SELECT} \ \overrightarrow{e \ \textsf{AS} \ l} \ \textsf{FROM} \ \overrightarrow{t \ \textsf{AS} \ y} \ \textsf{WHERE} \ B) \\
S[\![\textsf{where} \ B \ Z]\!] & = & \textsf{SELECT} \ \overrightarrow{e \ \textsf{AS} \ l} \ \textsf{FROM} \ \overrightarrow{t} \ \textsf{WHERE} \ S[\![B]\!] \wedge B' \\
 & & where \ S[\![Z]\!] \ = \ (\textsf{SELECT} \ \overrightarrow{e \ \textsf{AS} \ l} \ \textsf{FROM} \ \overrightarrow{t} \ \textsf{WHERE} \ B') \\
S[\![\textsf{table}(s)]\!] & = & \textsf{SELECT} \ \overrightarrow{s.l \ \textsf{AS} \ l} \ \textsf{FROM} \ s \ \textsf{WHERE TRUE} \\
S[\![\textsf{yield} \ R]\!] & = & \textsf{SELECT} \ S[\![R]\!] \ \textsf{FROM} \ \phi \ \textsf{WHERE TRUE} \\
S[\![[\langle \overrightarrow{l = B} \rangle]]\!] & = & \overrightarrow{S[\![B]\!] \ \textsf{AS} \ l} \\
S[\![\textsf{exists} \ U]\!] & = & \textsf{EXISTS} \ (S[\![U]\!]) \\
S[\![\oplus(\overrightarrow{B})]\!] & = & \oplus_{sql}(S[\![\overrightarrow{B}]\!]) \\
S[\![x.l]\!] & = & x.l \\
S[\![x]\!] & = & x.* \\
S[\![c]\!] & = & c
\end{array}
$$

図 3.5　SQL 変換

第 4 章

グループ化機能の拡張

本研究で新たに提案する、グループ化機能を拡張した QueΛ の構文を図 4.1 に示す。

Grouping key	k			
Terms	L, M, N	::=	...	
		|	**gfor** $(z \leftarrow \overrightarrow{k}\ (\overrightarrow{y \leftarrow M})\ L)\ N$	(grouping)
		|	**having** $L\ M$	(grouped test)
		|	**gyield** M	(grouped singleton)
		|	$\circledcirc(M)$	(aggregation function)

図 4.1　追加したグループ化機能の構文

gfor $(z \leftarrow \overrightarrow{k}\ (\overrightarrow{y \leftarrow M})\ L)\ N$ は、グループ化の列を指定するキー $\overrightarrow{k}$ に基づいて、グループ化を行う構成子である。L の部分には **where** 項での条件式と同等の働きをするものが入り、グループ化をする機能を除けば、テーブルから要素を取り出し評価を進めるという点で **for** 項と同じ働きをする。評価の手順としては、M から取り出した要素を y に束縛し、その環境の下で L が **true** のもののみ $\overrightarrow{k}$ によってグループ化する。そして、グループ分けされたバッグのレコードを変数 z に束縛し、その環境の下で項 N を評価していく。ここで、**for** 項は複数のテーブルを並べる際、**for** 項を連続して並べることで表現していたが、**gfor** 項で同じように表現しようとすると、正しく意味付けがされない。そこで、複数のテーブルを記述する際は $(y \leftarrow M)$ 部分で並べ、列として表現することで記述可能とした。項 N を評価した結果はレコードのバッグであり、それが **gfor** 項の結果となる。なお、**gfor** 項での変数 z は、SQL において登場しない変数となっているので SQL に変換するときには適切なレコードの変数として変換する。**having** $L\ M$ は、SQL での **HAVING** 句と同じ働きをし、基本的には **where** 項と同じ働きをするが、L には集約関数によって集約された列のみを許し、集約されていない列は次節の型付け規則によって制約され、記述できないようになっている。**gyield** M も M には集約された列のみを許し、それ以外は **yield** 項と同じ働きをする。$\circledcirc$ は集約関数を表し、SQL における値の平均値を算出する **AVG**、値の最大値を算出する **MAX** 関数や最小値を算出する **MIN** 関数等がある。

ここで、グループ化を行う SQL クエリと、それに対応するグループ化機能付き QueΛ のコードを図 4.2 に示す。

(SQL)
SELECT e.date **AS** date, c.class **AS** class, **MAX**(e.results) **AS** max
FROM en_test **AS** e, class **AS** c
WHERE e.sid $=$ c.sid
GROUP BY c.class, e.date

(QueΛ)
gfor $(z \leftarrow ((\text{date}, \text{class})\ ((e \leftarrow \textbf{table}(\text{“en\_test”})), (c \leftarrow \textbf{table}(\text{“class”})))\ (e.\text{sid} = c.\text{sid}))$
gyield $\langle$date $= z$.date,
class $= z$.class,
max $=$ **max**$(z$.results$)\rangle)$

図 4.2　グループ化を行う SQL クエリとそれに対応する QueΛ

図 4.2 では、2.1 節におけるテーブルに対してグループ化の処理を行っているが、ここで示すように、**gfor** の提案によって SQL クエリの **GROUP BY** だけでなく、テーブルが複数の時やクエリ中に **WHERE** 構文が存在する時にもサブクエリを使わずに直接対応するクエリの記述を書くことに成功した。

4.1　型付け規則の拡張

本節では新たに追加する **gfor** 等の型付け規則について記述していく。型付け規則を図 4.3 に示す。SQL において、**GROUP BY** 句でグループ化したクエリに対する **SELECT** 句や **HAVING** 句では、集約された列のみを許すという制約がある。そのため、新たに追加する型付け規則では、既存の型判断 $\Gamma \vdash M : A$ に加え、集約された項に対する型判断 $\Gamma \vdash_a M : A$、グループ化した後に記述できる構文に対する $\Gamma \vdash_G M : A$ を追加する。Σ は定数 c、プリミティブ演算子 $\oplus$、集約関数 $\circledcirc$ からそれらの型への写像である。グループ化の手順としては、**gfor** $(z \leftarrow \overrightarrow{k}\ (\overrightarrow{y \leftarrow M})\ L)\ N$ にて、最初に GFOR 規則が適用される。$\overrightarrow{k}$ は、$k_i \in \overrightarrow{l}$ より、型 A を持つレコードのフィールド l_i のいずれかが指定される。項 M はレコードのバッグであり、その要素を変数 y に束縛し、その環境の下で項 L を評価する。項 L が **true** であれば、$\overrightarrow{k}$ によってグループ化し、変数 z に束縛する。z はグループ化するためのキーと、同一のキーを持つ複数のレコードのバッグ $\langle \overrightarrow{k}, \text{Bag } A \rangle$ を持つバッグを型として持つ。そして、この環境の下で項 N を評価する。N 項は型判断 $\Gamma \vdash_G M : A$ によって導出され、HAVING、SINGLETONG、EMPTYG のいずれかの規則が適用され、**having** $L\ M$、**gyield** M、$[\,]$ のいずれかを記述することができる。

$\boxed{\Gamma \vdash M : A}$

GFOR
$$\frac{\Gamma \vdash M : \mathsf{Bag}\ \langle \overrightarrow{l : A} \rangle \quad \Gamma, y : \overrightarrow{\langle l : A \rangle} \vdash L : Bool \quad \Gamma, z : \langle \overrightarrow{k}, \mathsf{Bag}\ \langle \overrightarrow{l : A} \rangle \rangle \vdash_G N : \mathsf{Bag}\ B \quad k_i \in \overrightarrow{l}\ (\text{for each } 1 \leq i \leq n)}{\Gamma \vdash \mathbf{gfor}\ (z \leftarrow \overrightarrow{k}\ (y \leftarrow \overrightarrow{M})\ L)\ N : \mathsf{Bag}\ B}$$

$\boxed{\Gamma \vdash_a M : A}$

CONSTA
$$\frac{\Sigma(c) = O}{\Gamma \vdash_a c : O}$$

OPA
$$\frac{\Sigma(\oplus) = O_1 \times ... \times O_n \to O \quad \Gamma \vdash_a M_i : O_i\ (\text{for each } 1 \leq i \leq n)}{\Gamma \vdash_a \oplus(\overrightarrow{M}) : O}$$

RECORDA
$$\frac{\Gamma \vdash_a M_i : A_i\ (\text{for each } 1 \leq i \leq n)}{\Gamma \vdash_a \langle \overrightarrow{l = M} \rangle : \langle \overrightarrow{l : A} \rangle}$$

PROJECTA
$$\frac{\Gamma \vdash_a L : \langle \overrightarrow{l : A} \rangle}{\Gamma \vdash_a L.l_i : A_i}$$

PROJECTK
$$\frac{z : \langle \overrightarrow{k}, \mathsf{Bag}\ \langle \overrightarrow{l : A} \rangle \rangle \in \Gamma}{\Gamma \vdash_a z.k_i : A_i}$$

AGGREGATION
$$\frac{\Sigma(\circledcirc) = O_1 \to O_2 \quad \Gamma \vdash M : O_1}{\Gamma \vdash_a \circledcirc(M) : O_2}$$

$\boxed{\Gamma \vdash_G M : A}$

HAVING
$$\frac{\Gamma \vdash_a L : Bool \quad \Gamma \vdash_G M : \mathsf{Bag}\ A}{\Gamma \vdash_G \mathbf{having}\ L\ M : \mathsf{Bag}\ A}$$

SINGLETONG
$$\frac{\Gamma \vdash_a M : A}{\Gamma \vdash_G \mathbf{gyield}\ M : \mathsf{Bag}\ A}$$

EMPTYG
$$\frac{}{\Gamma \vdash_G [\,] : \mathsf{Bag}\ A}$$

図 4.3　グループ化機能を追加した QueΛ の型付け規則

4.2　グループ化機能を追加した QueΛ の意味論

本節では、グループ化で必要不可欠な集約関数の動きについて記述した後、前節での型付け規則に基づいた **gfor** 等の評価文脈、操作的意味論について記述する。

4.2.1　集約関数

SQL において、**GROUP BY** 句を用いて最終的に返せる評価結果の一つが、集約関数を用いて集約された列である。集約関数には、前にも述べたように SQL における **AVG** 関数、**MAX** 関数、**MIN** 関数等が存在し、QueΛ では **avg**、**max**、**min** 等として記述する。本研究において、これらの集約関数の多くはそのまま表現することができるが、一部で例外が存在する。例えば、以下のようなクエリに対する場合である。

$$\mathbf{gfor}\ (z \leftarrow \overrightarrow{k}\ (y \leftarrow M \uplus N)\ L)\ (\mathbf{avg}(S))$$

このクエリは、それぞれのテーブルを内部結合したテーブルに対して、$\overrightarrow{k}$ でグループ化した上で、S の平均値

を求めるという操作である。このクエリを SQL で直接表現するには、以下のように $\uplus$ を外側に出し、最後にそれぞれのクエリを結合させる形にしなければならない。

$$\textbf{gfor}\ (z \leftarrow \overrightarrow{k}\ (y \leftarrow M)\ L)\ S' \uplus \textbf{gfor}\ (z \leftarrow \overrightarrow{k}\ (y \leftarrow N)\ L)\ S'$$

しかし、この変換において等価なクエリを表現しようとすると、それぞれのクエリで合計値とデータの数を求め、2 つのテーブルの合計値の和をデータの数の和で割ることでしか平均値を求めることはできない。すなわち、変換の前後で異なる集約関数を使わなければならない。そこで本研究では、この変換を可能にするために $norm(\circledcirc(S))$ といったマクロを用意し、正規化の前に必要に応じて集約関数を書き換える。しかし、中央値を求めるような集約関数は全データを必要とし、$norm$ を用いても表すことができないため、本研究ではこのような統計的な集約関数を制限して、代数的な集約関数のみを使えるようにする。

4.2.2 操作的意味論の拡張

意味論において、グループ化に対する解釈を必要とするため、レコードのバッグに対し、グループ化直後のレコードのバッグを $[\langle \mathsf{key} = k.., \mathsf{body} = [x..]\rangle]$ として内部的に用意し、以下のような補助関数 ($sub\_group, group, sub\_eval\_group, eval\_group$) を与える。これらに対する定義を図 4.4 に示す。

sub_group

グループ化するためのキー ($\overrightarrow{k}$)、レコード (x)、グループ化した途中段階のレコードのバッグ (N) を引数にとる。N の要素に対して、$x.k$ が一致すれば、グループ化後のバッグの body 部分にレコード x を加え、一致しなければ新たなレコードとしてバッグに加える。

group

グループ化するためのキー ($\overrightarrow{k}$)、レコードのバッグとその要素を束縛した変数 ($(y \leftarrow V)$)、条件式 (L) を引数にとる。L が **true** であれば要素 x を sub_group に渡し、グループ化する。最終的にすべての要素に対してグループ化したバッグが返される。

sub_eval_group

グループ化したバッグ (G)、グループ化後評価する項 (M) を引数にとる。グループ化したバッグのそれぞれの要素に対して項 M を評価し、集約関数等を用いたフラットな構造のレコードのバッグが返される。

eval_group

グループ化するためのキー ($\overrightarrow{k}$)、レコードのバッグとその要素を束縛した変数 ($(y \leftarrow V)$)、条件式 (L)、グループ化後評価する項 (M) を引数にとる。$\overrightarrow{k}, (x \leftarrow V), L$ を $group$ に渡し、グループ化したバッグの要素に対して項 M を評価する。

$sub\_group(\overrightarrow{k}, v, N)$ =
$N = [\,]$ のとき $[\langle \mathsf{key} = v.\overrightarrow{k}, \mathsf{body} = [v] \rangle]$
$N = n :: N'$ のとき if $n.\mathsf{key} = v.\overrightarrow{k}$ then $[\langle \mathsf{key} = n.\mathsf{key}, \mathsf{body} = v :: n.\mathsf{body} \rangle] @ N'$
else $n :: sub\_group(\overrightarrow{k}, v, N')$

$group(\overrightarrow{k}, (y \leftarrow V), L)$ =
$V = [\,]$ のとき $[\,]$
$V = v :: V'$ のとき let $gr = group(\overrightarrow{k}, (y \leftarrow V'), L)$ in
if $L[v/y]$ then $sub\_group(\overrightarrow{k}, v, gr)$
else gr

$sub\_eval\_group(G, M)$ =
$G = [\,]$ のとき $[\,]$
$G = g :: G'$ のとき $M @ sub\_eval\_group(G', M)$

$eval\_group(\overrightarrow{k}, (y \leftarrow V), L, M)$ =
let $g = group(\overrightarrow{k}, (y \leftarrow V), L)$ in $sub\_eval\_group(g, M)$

図 4.4　グループ化に対する新たな補助関数

新たに追加する評価文脈、操作的意味論を図 4.5 に示す。評価文脈には、集約関数、**gfor**、**having**、**gyield** が追加され、**gfor** においては、テーブルの要素を取り出してから、条件分岐をするという評価順序となっている。操作的意味論には、集約関数、**having**、**gfor** の意味論が追加されている。**gfor** に関して E-GFOR で記述されている $sub\_eval\_group, group, sub\_group$ については、図 4.4 のプログラム形式で書かれている補助関数を意味論の形式で定義し直したものである。集約関数は最初に前節の $norm$ によって正規化され、その環境の下で **gfor** 項の意味論が定義されている。また、ここでの $M[g.\mathsf{body}/z]$ では、項 M において z が存在すれば、g.body を代入する操作を行っている。例えば、以下のような全クラスのテスト結果の平均点を求める M と、それぞれのテスト結果、クラス情報を持つ g.body に対し、$M[g.\mathsf{body}/z]$ では項 M の z に g.body を代入し、全てのテストの点数の合計点をクラス数で割ることで平均点を求める操作をしている。ここでは集約関数が正規化されているため、平均点を求める操作が **sum** 関数と **count** 関数で表されている。

$$
\begin{aligned}
M &= [\langle \mathsf{avg} = \mathbf{sum}(z.\mathsf{results})/\mathbf{count}(z.\mathsf{class}) \rangle] \\
g.\mathsf{body} &= [\langle \mathsf{results} = 40, \mathsf{class} = \text{“}A\text{”} \rangle; \langle \mathsf{results} = 60, \mathsf{class} = \text{“}B\text{”} \rangle] \\
M[g.\mathsf{body}/z] &= [\langle \mathsf{avg} = \mathbf{sum}([40; 60])/\mathbf{count}([\text{“}A\text{”}; \text{“}B\text{”}]) \rangle] \\
&= [\langle \mathsf{avg} = 50 \rangle]
\end{aligned}
$$

Value and Evaluation Contexts

$$
\begin{array}{lrl}
\text{Evaluation Context} & \mathcal{E} & ::= \quad ... \\
& | & \circledcirc(\overrightarrow{V}, \mathcal{E}, \overrightarrow{M}) \;\mid\; \textsf{gfor}\ (z \leftarrow \overrightarrow{k}\ ((\overrightarrow{y_1 \leftarrow V}), (y'_1 \leftarrow \mathcal{E}), (\overrightarrow{y''_1 \leftarrow L}))\ M)\ N \\
& | & \textsf{gfor}\ (z \leftarrow \overrightarrow{k}\ (\overrightarrow{y_1 \leftarrow V})\ \mathcal{E})\ N \;\mid\; \textsf{having}\ \mathcal{E}\ M \;\mid\; \textsf{gyield}\ \mathcal{E}
\end{array}
$$

Operational Semantics

...

$$
\begin{array}{rcll}
\circledcirc(\overrightarrow{V}) & \longrightarrow & \sigma(\circledcirc, \overrightarrow{V}) & \text{(E-AGGREGATION)} \\
\textsf{having true}\ M & \longrightarrow & M & \text{(E-HAVINGTRUE)} \\
\textsf{having false}\ M & \longrightarrow & [\,] & \text{(E-HAVINGFALSE)} \\
\textsf{gfor}\ (z \leftarrow \overrightarrow{k}\ (y \leftarrow [\,])\ L)\ N & \longrightarrow & [\,] & \text{(E-GFOREMPTY}_1\text{)} \\
\textsf{gfor}\ (z \leftarrow \overrightarrow{k}\ (y \leftarrow M)\ L)\ N & \longrightarrow & & \\
\multicolumn{4}{r}{sub\_eval\_group(z \leftarrow group(\overrightarrow{k}, (y \leftarrow V), L), M)\ \text{(E-GFOR)}}
\end{array}
$$

$$
\begin{array}{lrll}
\textcircled{1} & sub\_eval\_group((z \leftarrow G), M) & \rightarrow & [\,]\ (\textsf{if}\ G = [\,]) \\
& sub\_eval\_group((z \leftarrow G), M) & \rightarrow & \\
& \multicolumn{3}{r}{M[g.\textsf{body}/z] :: sub\_eval\_group((z \leftarrow G'), M)\ (\textsf{if}\ G = g :: G')} \\
\textcircled{2} & group(\overrightarrow{k}, (y \leftarrow V), L) & \rightarrow & [\,]\ (\textsf{if}\ V = [\,]) \\
& group(\overrightarrow{k}, (y \leftarrow V), L) & \rightarrow & sub\_group(\overrightarrow{k}, v, group(k, (y \leftarrow V), L)) \\
& & & (\textsf{if}\ V = v :: V' \wedge L[v/y] \rightarrow \textbf{true}) \\
& group(\overrightarrow{k}, (y \leftarrow V), L) & \rightarrow & group(\overrightarrow{k}, (y \leftarrow V'), L) \\
& & & (\textsf{if}\ V = v :: V' \wedge L[v/y] \rightarrow \textbf{false}) \\
\textcircled{3} & sub\_group(\overrightarrow{k}, v, N) & \rightarrow & [\langle\, \textsf{key} = v.\overrightarrow{k},\ \textsf{body} = [v]\, \rangle]\ (\textsf{if}\ N = [\,]) \\
& sub\_group(\overrightarrow{k}, v, N) & \rightarrow & [\langle\, \textsf{key} = n.\textsf{key},\ \textsf{body} = v :: n.\textsf{body}\, \rangle] \mathbin{@} N' \\
& & & (\textsf{if}\ N = n :: N' \wedge n.\textsf{key} = v.\overrightarrow{k}) \\
& sub\_group(\overrightarrow{k}, v, N) & \rightarrow & n :: sub\_group(\overrightarrow{k}, v, N') \\
& & & (\textsf{if}\ N = n :: N' \wedge n.\textsf{key} \neq v.\overrightarrow{k})
\end{array}
$$

図 4.5　グループ化機能を追加した QueΛ における評価文脈, 操作的意味論の定義

4.3 拡張する正規形と正規化規則

グループ化機能を拡張するために QueΛ にいくつかの構成子を新たに追加した。本節では QueΛ によって書かれたグループ化のクエリを SQL に変換するための正規形と正規化規則を記述する。

4.3.1 正規形

SQL の構文と一対一に対応する統合言語クエリの構文を正規形と呼び、既存の正規形と新たに追加する正規形を図 4.6 に示す。

Queries	U	::=	$U_1 \uplus U_2 \mid [\,] \mid F \mid F_g$
Comprehension	F	::=	$\mathbf{for}(x \leftarrow H)\ F \mid Z$
	F_g	::=	$\mathbf{gfor}\ (z \leftarrow \overrightarrow{K_g}\ (\overrightarrow{y_1 \leftarrow H})\ B)\ Z_g$
	H	::=	$F_g \mid \mathbf{table}(t)$
Body	Z	::=	$\mathbf{where}\ B\ Z \mid \mathbf{yield}\ R$
	Z_g	::=	$\mathbf{having}\ B_g\ Z_g \mid \mathbf{gyield}\ R_g$
Record	R	::=	$\langle \overrightarrow{l = B} \rangle \mid x$
	R_g	::=	$\langle \overrightarrow{l = B_g} \rangle$
Primitives	K_g	::=	k
	B	::=	$\mathbf{exists}\ U \mid \oplus(\overrightarrow{B}) \mid x.l \mid y.l \mid c \mid z.l$
Aggregation	B_g	::=	$\circledcirc(\overrightarrow{B}) \mid c \mid z.k \mid \oplus(\overrightarrow{B_g})$

図 4.6　グループ化機能を追加した QueΛ の正規形

グループ化をすることが先行研究と異なる点は、ネストされた構文に対して、コードを最適化することができない場合があるという点である。これまでの先行研究では、**for** 項のテーブル部分がネストされた構造になっていれば、正規化規則によりネストがないフラットな構造にコードを書き換えられたのに対し、例えば、グループ化したものをさらにグループ化するという操作においては、ネストされた構造と同じ意味を持つフラットな構造な SQL クエリは存在しない。そのため、ネストされたものが最小のクエリとなり、グループ化に対し、ネストされた構造を正規形として許さなければならない。図 4.6 では、**gfor** 項と **table** 項を内包表記 H にまとめ、**for** 項、**gfor** 項のテーブル部分に書けるようにした。**gfor** 項を用いて複数のテーブルをグループ化するクエリについては、テーブルの要素を変数に束縛する操作を $(\overrightarrow{y \leftarrow H})$ 部分に列として並べて表現するため、グループ化付き SQL クエリに対応する **gfor** 項は常に一つとなる。本研究では、**for** 項と **gfor** 項をどのように対応付けするかが肝になっている。そこで、**for** 項と **gfor** 項の関係をわかりやすく記述したものを表 4.1 に示す。**for** 項同士の組み合わせは先行研究 [2] における正規形と正規化規則を用いて表現することができる。**gfor** 項と **for** 項が連続して書かれるクエリは、グループ化した途中段階のテーブルのそれぞれの要素に対して、**for** 項を評価するという意味になる。グループ化した途中段階のテーブルに対する操作というのは、前節で記述したような key と body を持つレコードのバッグに対しての操作ということであり、型が合わないだけでなく、記述されたクエリに対して適切な意味付

けがされないことからそういったクエリを記述することはできない。**gfor** 項と **gfor** 項が連続して書かれるクエリに対しても同様の理由で適切な意味付けをすることができない。そのため、**gfor** のボディー部分に記述することができる項は **having** 項と **gyield** 項の 2 つのみであり、これを Z_g として表記する。**having** 項の条件式、もしくは **gyield** 項のボディーとして書けるものは定数、グループ化したキー、集約関数を用いて評価された値であり、これらを R_g として表記する。本研究では表 4.1 内の下の二つに対して注目し、次項で述べる正規化規則を用いることで正規化することに成功した。

表 4.1　**for** 項と **gfor** 項の関係

for 項と **gfor** 項の組み合わせ	意味
for $(x \leftarrow$ **for** ...$)$	(先行研究での) 正規化規則により書き換え可
for $(x \leftarrow M)$ **for** ...	正規形
gfor $(z \leftarrow$ ...$)$ **for** ...	適切な意味付けがされない
gfor $(z \leftarrow$ ...$)$ **gfor** ...	適切な意味付けがされない
for $(x \leftarrow$ **gfor** ...$)$	サブクエリとして記述
gfor $(z \leftarrow (\overrightarrow{k}\ (y \leftarrow$ **gfor** ...$)))$	サブクエリとして記述
gfor $(z \leftarrow (\overrightarrow{k}\ (y \leftarrow$ **for** ...$)))$	(本研究での) 正規化規則により書き換え可
for $(x \leftarrow M)$ **gfor** ...	(本研究での) 正規化規則により書き換え可

4.3.2　正規化規則

新たに追加する正規化規則を図 4.7 に示す。参考のため、全体の正規化規則は付録 A に記載する。

Stage 1 では、ネストされている中間のデータ構造に対して、グループ化せずに書かれているクエリがあれば、フラットなデータ構造に書き換える操作をしている。GFORFOR 規則、GFORWHERE 規則、GFORYIELD 規則では、ネストされている **for** 項、**where** 項、**yield** 項に対して、**gfor** 項のテーブル、条件式、ボディーの一部として書き換えることでフラットなデータ構造にしている。本研究では、**gfor** 項のテーブル部分で、レコードを束縛する処理を列として並べて記述したことにより、複数のテーブルに対する処理を可能にした。この処理によって GFORFOR 規則の書き換えは可能になっており、条件式を **gfor** 項の中に埋め込んだことで、GFORWHERE 規則の書き換えを可能にした。

Stage 2 では、クエリを意味的に同じ SQL クエリに直接変換できるように並べかえる操作を行なっている。FORGFOR 規則は、**for** 項のレコードそれぞれに対して **gfor** 項の要素をグループ化するという意味であり、これを SQL に直接変換できるように書き換えると、2 つのテーブルそれぞれの要素に対し、$\overrightarrow{k}, x$ のレコード全体でグループ化するというものと同義である。故に一つの **gfor** 項にまとめることができ、同じ意味を持つ SQL クエリに直接変換できる。しかしこのとき、レコード y とレコード x のフィールド名に同一のものが存在していれば、判別するためにフィールド名を書き換えなければならない。そのため、正規化によってフィールド名を書き換える関数 f_{re} を以下のように用意する。

$$f_{re}\, x \;=\; \langle\ \overrightarrow{l'} \;=\; x.\overrightarrow{l}\ \rangle$$

関数 f_{re} ではレコード x を引数にとり、レコードのそれぞれのフィールドにダッシュを付け、そのレコードを

返す操作をしている。この操作により同じフィールド名に対しての判別を可能にしている。また、関数 $FN(m)$ ではレコード m のフィールド名を取り出す操作をしている。WHEREFOR 規則において、グループ化したバッグの要素に対し、条件式で制御するというのは、グループ化後に **having** を使うことと同義であるため、図 4.7 のように書き換えることができる。

(**Stage 1**)

$$\textbf{gfor}\ (z \leftarrow \overrightarrow{k}\ (y \leftarrow \textbf{for}\ (x \leftarrow M)\ N)\ L)\ O \quad \rightsquigarrow \quad \textbf{gfor}\ (z \leftarrow \overrightarrow{k}\ ((x \leftarrow M), (y \leftarrow N))\ L)\ O \qquad \text{(GFORFOR)}$$

$$\textbf{gfor}\ (z \leftarrow \overrightarrow{k}\ (y \leftarrow \textbf{where}\ M\ N)\ L)\ O \quad \rightsquigarrow \quad \textbf{gfor}\ (z \leftarrow \overrightarrow{k}\ (y \leftarrow N)\ (L \wedge M))\ O \qquad \text{(GFORWHERE)}$$

$$\textbf{gfor}\ (z \leftarrow \overrightarrow{k}\ (y \leftarrow \textbf{yield}\ M)\ L)\ N \quad \rightsquigarrow \quad \textbf{where}\ L\ (N[y := M]) \qquad \text{(GFORYIELD)}$$

$$\textbf{gfor}\ (z \leftarrow \overrightarrow{k}\ (y \leftarrow [\,]\ )\ L)\ N \quad \rightsquigarrow \quad [\,] \qquad \text{(GFOREMPTY}_1\text{)}$$

$$\textbf{having true}\ M \quad \rightsquigarrow \quad M \qquad \text{(HAVINGTRUE)}$$

$$\textbf{having false}\ M \quad \rightsquigarrow \quad [\,] \qquad \text{(HAVINGFALSE)}$$

(**Stage 2**)

$$\textbf{for}\ (x \leftarrow M)\ \textbf{gfor}\ (z \leftarrow \overrightarrow{k}\ (\overrightarrow{y \leftarrow N})\ L)\ O \quad \hookrightarrow \quad \textbf{gfor}\ (z \leftarrow (\overrightarrow{k}, \overrightarrow{l'})\ ((x \leftarrow M), (\overrightarrow{y \leftarrow N}))\ L)\ O \quad (where\ m' = f_{re}\ (m :: M),\ l' = FN(m')) \qquad \text{(FORGFOR)}$$

$$\textbf{where}\ L\ \textbf{gfor}\ (\overrightarrow{z} \leftarrow \overrightarrow{k}\ (\overrightarrow{y \leftarrow N})\ M)\ O \quad \hookrightarrow \quad \textbf{gfor}\ (z \leftarrow \overrightarrow{k}\ (\overrightarrow{y \leftarrow N})\ M)\ \textbf{having}\ L\ O \qquad \text{(WHEREGFOR)}$$

$$\textbf{gfor}\ (z \leftarrow \overrightarrow{k}\ (\overrightarrow{y \leftarrow M})\ L)\ [\,] \quad \hookrightarrow \quad [\,] \qquad \text{(GFOREMPTY}_2\text{)}$$

$$\textbf{having}\ L\ [\,] \quad \hookrightarrow \quad [\,] \qquad \text{(HAVINGEMPTY)}$$

$$\textbf{having}\ L\ (\textbf{having}\ M\ N) \quad \hookrightarrow \quad \textbf{having}\ (L \wedge M)\ N \qquad \text{(HAVINGHAVING)}$$

図 4.7　追加する正規化規則

4.4　SQL 変換の拡張

図 4.6 の正規形に対応する SQL サブ言語と、SQL へ変換するための変換規則を図 4.8、図 4.9 に示す。SQL における **GROUP BY** 句, **HAVING** 句を追加しただけでなく、グループ化のキーを表す $y.k$ や集約関数を表す $\circledcirc_{sql}$ を追加した。QueΛ において、**gfor** 項における変数 z は、元のレコードをグループ化した中間的なレコードであり、変数 z に直接対応する SQL クエリは存在しないため、z が扱うフィールド名によって変数を変更して変換している。

$$
\begin{array}{rcl}
P, Q & := & P \ \textsf{UNION ALL} \ Q \mid S \\
S & := & \textsf{SELECT} \ \overrightarrow{s} \ \textsf{FROM} \ t \ \textsf{AS} \ x \ \textsf{WHERE} \ e \\
T & := & \textsf{SELECT} \ \overrightarrow{u} \ \textsf{FROM} \ t \ \textsf{AS} \ y \ \textsf{WHERE} \ e \ \textsf{GROUP BY} \ \overrightarrow{k} \ \textsf{HAVING} \ f \\
s & := & e \ \textsf{AS} \ l \mid x.* \\
u & := & f \ \textsf{AS} \ l \\
e & := & c \mid x.l \mid e \wedge e' \mid \neg e \mid \textsf{EXISTS}(Q) \mid \oplus_{sql}(\overrightarrow{e}) \mid y.l \\
f & := & c \mid y.k \mid f \wedge f' \mid \neg f \mid \oplus_{sql}(\overrightarrow{f}) \mid \circledcirc_{sql}(\overrightarrow{e})
\end{array}
$$

図 4.8　グループ化機能を追加した SQL サブ言語

$$
\begin{array}{rcl}
 & & \ldots \\
\mathrm{S}[\![\textsf{gfor} \ (z \leftarrow \overrightarrow{K_g} \ (\overrightarrow{y \leftarrow H}) \ B) \ Z_g]\!] & = & \textsf{SELECT} \ \overrightarrow{d \ \textsf{AS} \ l} \ \textsf{FROM} \ \overrightarrow{(S[\![H]\!]) \ \textsf{AS} \ y}, \overrightarrow{t \ \textsf{AS} \ h} \ \textsf{WHERE} \ S[\![B]\!] \ \textsf{AND} \ B' \\
 & & \textsf{GROUP BY} \ S[\![\overrightarrow{K_g}]\!], \overrightarrow{k'} \ \textsf{HAVING} \ B_g \\
where \ S[\![Z_g]\!] & = & (\textsf{SELECT} \ \overrightarrow{d \ \textsf{AS} \ l} \ \textsf{FROM} \ \overrightarrow{t \ \textsf{AS} \ h} \ \textsf{WHERE} \ B' \ \textsf{GROUP BY} \ \overrightarrow{k'} \ \textsf{HAVING} \ B_g) \\
\mathrm{S}[\![\textsf{having} \ B_g \ Z_g]\!] & = & \textsf{SELECT} \ \overrightarrow{d \ \textsf{AS} \ l} \ \textsf{FROM} \ \overrightarrow{t} \ \textsf{WHERE} \ B \\
 & & \textsf{GROUP BY} \ \overrightarrow{k} \ \textsf{HAVING} \ S[\![B_g]\!] \ \textsf{AND} \ B'_g \\
where \ S[\![Z_g]\!] & = & (\textsf{SELECT} \ \overrightarrow{d \ \textsf{AS} \ l} \ \textsf{FROM} \ \overrightarrow{t} \ \textsf{WHERE} \ B \ \textsf{GROUP BY} \ \overrightarrow{k} \ \textsf{HAVING} \ B'_g) \\
\mathrm{S}[\![\textsf{gyield} \ R_g]\!] & = & \textsf{SELECT} \ S[\![R_g]\!] \ \textsf{FROM} \ \phi \ \textsf{WHERE TRUE} \\
 & & \textsf{GROUP BY} \ \phi \ \textsf{HAVING TRUE} \\
\mathrm{S}[\![[\langle \overrightarrow{l = B_g} \rangle]]\!] & = & \overrightarrow{S[\![B_g]\!] \ \textsf{AS} \ l} \\
\mathrm{S}[\![\oplus(\overrightarrow{B_g})]\!] & = & \oplus_{sql}(S[\![\overrightarrow{B_g}]\!]) \\
\mathrm{S}[\![\circledcirc(\overrightarrow{B})]\!] & = & \circledcirc_{sql}(S[\![\overrightarrow{B}]\!]) \\
\mathrm{S}[\![y.l]\!] & = & y.l \\
\mathrm{S}[\![z.l]\!] & = & y.l \quad where \ l = FN(y) \\
\mathrm{S}[\![z.k]\!] & = & y.k \\
\mathrm{S}[\![\overrightarrow{k}]\!] & = & \overrightarrow{k}
\end{array}
$$

図 4.9　SQL 変換

4.5 正規化、SQL 変換の例

前節までの提案に対し、正規化、SQL 変換がどのように行われるかを例題を見ながら概説していく。ここでは、2.1 節のテーブルに対し、2 つのクエリ (Q_1, Q_4) を用意し、*compose* 関数によって 2 つのクエリを合成し、正規化によって単一の SQL クエリに変換するという操作を行う。クエリ Q_1 は、2.1 節におけるクエリと同じものであり、class テーブルから与えられたクラスを取り出す操作をするクエリである。クエリ Q_4 は、en_test テーブルから Q_4 の生徒と同一の生徒を取り出し、その生徒の英語の平均点を求める操作をするクエリである。新たなクエリ Q_4 を以下に示す。

$$
\begin{array}{l}
Q_4 \ = \ \lambda c. \\
\quad \textbf{gfor} \ (z \ \leftarrow \ ((\mathsf{sid}, \mathsf{class}) \ (e \leftarrow \textbf{table}(\text{“en\_test”})) \ (e.\mathsf{sid} \ = \ c.\mathsf{sid})) \\
\quad \textbf{gyield} \ \langle \ \mathsf{sid} \ = \ z.\mathsf{sid}, \\
\qquad\qquad \mathsf{class} \ = \ c.\mathsf{class}, \\
\qquad\qquad \mathsf{avg} \ = \ \textbf{avg}(z.\mathsf{results}) \ \rangle
\end{array}
$$

compose 関数を用いて 2 つのクエリ Q_1, Q_4 を合成したクエリを Q_5 とし、展開したものを以下に示す。

$$
\begin{array}{l}
Q_5 \ = \ \lambda x. \\
\quad \textbf{for} \ (y \ \leftarrow \ (\lambda class. \\
\qquad\qquad \textbf{for} \ (c \leftarrow \textbf{table}(\text{“class”})) \\
\qquad\qquad \textbf{where} \ (c.\mathsf{class} \ = \ class) \\
\qquad\qquad \textbf{yield} \ c) \ x) \\
\quad (\lambda c. \\
\quad \textbf{gfor} \ (z \ \leftarrow \ ((\mathsf{sid}, \mathsf{class}) \ (e \leftarrow \textbf{table}(\text{“en\_test”})) \ (e.\mathsf{sid} \ = \ c.\mathsf{sid})) \\
\quad \textbf{gyield} \ \langle \ \mathsf{sid} \ = \ z.\mathsf{sid}, \\
\qquad\qquad \mathsf{class} \ = \ c.\mathsf{class}, \\
\qquad\qquad \mathsf{avg} \ = \ \textbf{avg}(z.\mathsf{results}) \ \rangle) \ y
\end{array}
$$

まず、λ 抽象を簡約化するために、Q_5 に “A” を適用する。

$$
\begin{array}{l}
Q_5 \ = \ \textbf{for} \ (y \ \leftarrow \ (\textbf{for} \ (c \leftarrow \textbf{table}(\text{“class”})) \\
\qquad\qquad \textbf{where} \ (c.\mathsf{class} \ = \ \text{“}A\text{”}) \\
\qquad\qquad \textbf{yield} \ c)) \\
\quad (\lambda c. \\
\quad \textbf{gfor} \ (z \ \leftarrow \ ((\mathsf{sid}, \mathsf{class}) \ (e \leftarrow \textbf{table}(\text{“en\_test”})) \ (e.\mathsf{sid} \ = \ c.\mathsf{sid})) \\
\quad \textbf{gyield} \ \langle \ \mathsf{sid} \ = \ z.\mathsf{sid}, \\
\qquad\qquad \mathsf{class} \ = \ c.\mathsf{class}, \\
\qquad\qquad \mathsf{avg} \ = \ \textbf{avg}(z.\mathsf{results}) \ \rangle) \ y
\end{array}
$$

次に、**Stage 1** における FORFOR 規則、FORWHERE 規則、FORYIELD 規則を適用して、フラットなデータ構造に書き換える。

$$
\begin{array}{l}
Q_5 \ = \ \textbf{for} \ (c \leftarrow \textbf{table}(\text{“class”})) \\
\quad \textbf{where} \ (c.\mathsf{class} \ = \ \text{“}A\text{”}) \\
\quad \textbf{gfor} \ (z \ \leftarrow \ ((\mathsf{sid}, \mathsf{class}) \ (e \leftarrow \textbf{table}(\text{“en\_test”})) \ (e.\mathsf{sid} \ = \ c.\mathsf{sid})) \\
\quad \textbf{gyield} \ \langle \ \mathsf{sid} \ = \ z.\mathsf{sid}, \\
\qquad\qquad \mathsf{class} \ = \ c.\mathsf{class}, \\
\qquad\qquad \mathsf{avg} \ = \ \textbf{avg}(z.\mathsf{results}) \ \rangle
\end{array}
$$

次に、SQL に変換するために **Stage 2** の正規化規則を用いて正規化を行っていく。
まず、WHEREGFOR 規則を適用する。

$$
\begin{aligned}
Q_5 \;=\; & \textbf{for}\ (c \leftarrow \textbf{table}(\text{“class”})) \\
& \quad \textbf{gfor}\ (z \;\leftarrow\; ((\mathsf{sid}, \mathsf{class})\ (e \leftarrow \textbf{table}(\text{“en\_test”}))\ (e.\mathsf{sid} \;=\; c_1.\mathsf{sid})) \\
& \quad \textbf{having}\ (z.\mathsf{class} \;=\; \text{“}A\text{”}) \\
& \quad \textbf{gyield}\ \langle\, \mathsf{sid} \;=\; z.\mathsf{sid}, \\
& \qquad\qquad \mathsf{class} \;=\; c.\mathsf{class}, \\
& \qquad\qquad \mathsf{avg} \;=\; \textbf{avg}(z.\mathsf{results})\ \rangle
\end{aligned}
$$

次に、FORGFOR 規則を適用する。

$$
\begin{aligned}
Q_5 \;=\; & \textbf{gfor}\ (z \;\leftarrow\; ((\mathsf{sid}, \mathsf{class}, \mathsf{sid}', \mathsf{class}')\ ((e \leftarrow \textbf{table}(\text{“en\_test”})),\ (c \leftarrow \textbf{table}(\text{“class”})))\ (e.\mathsf{sid} \;=\; c.\mathsf{sid})) \\
& \quad \textbf{having}\ (z.\mathsf{class} \;=\; \text{“}A\text{”}) \\
& \quad \textbf{gyield}\ \langle\, \mathsf{sid} \;=\; z.\mathsf{sid}, \\
& \qquad\qquad \mathsf{class} \;=\; z.\mathsf{class}, \\
& \qquad\qquad \mathsf{avg} \;=\; \textbf{avg}(z.\mathsf{results})\ \rangle
\end{aligned}
$$

上記のように正規項が得られたので、これを SQL クエリに変換していく。
SQL 変換では、**gfor** $(z \;\leftarrow\; (\overrightarrow{k}\ (\overrightarrow{y \leftarrow M})\ L)\ N$ 項において、先に項 N から変換される。よって、最初に **gyield** 項を変換する。

```
SELECT e.sid AS sid, c.class AS class, AVG(e.results) AS avg
FROM φ
WHERE TRUE
GROUP BY φ
HAVING TRUE
```

次に、上記の結果を持って **having** 項を変換する。

```
SELECT e.sid AS sid, c.class AS class, AVG(e.results) AS avg
FROM φ
WHERE TRUE
GROUP BY φ
HAVING TRUE AND c.class = “A”
```

最後に、上記の結果を持って **gfor** 項を変換する。

```
SELECT e.sid AS sid, c.class AS class, AVG(e.results) AS avg
FROM en_test AS e, class AS c
WHERE TRUE AND e.sid = c.sid
GROUP BY e.sid, c.class, c
HAVING TRUE AND c.class = “A”
```

よって最終的な SQL クエリとして上記のクエリが得られ、グループ化機能を持つ QueΛ のクエリを正規化によって単一の SQL クエリに変換することができた。

第 5 章

結論

5.1 まとめ

本研究では、対象言語 QueΛ に対して、**gfor** や **having**、**gyield** などの構文を追加したことで、グループ化機能を持つ統合言語クエリを設計した。それぞれのレコードを変数に束縛する処理を列として並べ、複数のテーブルを表現するようにしただけでなく、グループ化の前に条件分岐を組み込むことで、グループ化機能付き SQL に直接対応する統合言語クエリの記述を可能にした。また、SQL における、グループ化の **UNION ALL** について、代数的に使える集約関数しか表現できないことを発見し、**AVG** 関数のような集約関数を書き換えることで正規化において意味を変えない等価なクエリとして表現することを可能にした。

本研究における正規化規則は、先行研究とは異なるアプローチにより追加されている。先行研究における正規化規則ではネストされた構造のクエリや SQL に直接変換できないクエリをただ並べ替えるような正規化規則を追加することで正規化を可能としてきた。しかし、本研究では並べ替えるだけでは適切な正規化がされず、4.3.2 における GFORFOR 規則や FORGFOR 規則のように複数のテーブルの処理やグループ化キーを増やす等、先行研究とは異なるやり方で正規化規則を追加することで正規化を可能とした。

5.2 今後の課題

本研究ではグループ化機能を持つ統合言語クエリを設計したが、処理系を用いた正規化規則等の型付け保存性はチェックされていない。これらの型付け保存性を証明するために、先行研究では OCaml での終タグレス法による実装を行っていたが [4]、グループ化後に生成されるレコードのバッグや集約関数の組み込みなどグループ化によって型が変わってしまうことを考えると、終タグレス法では表しきれないと考えられる。よって、別のアプローチを用いて本研究で提案した統合言語クエリの正しさを証明することが今後の課題としてあげられる。

謝辞

本研究を進めるにあたり、様々なご指導をしてくださった亀山幸義教授および海野広志准教授に深く感謝いたします。また、多くのご助言を下さったプログラム論理研究室の皆様に感謝を申し上げます。

参考文献

[1] Torsten Grust, Jan Rittinger, and Tom Schreiber. Avalanche-safe linq compilation. In *Proceedings of the VLDB Endowment*, Volume 3 Issue 1-2, pp. 162–172, September 2010.

[2] James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. In *Proceedings of 18th ACM SIGPLAN international conference on Functional programming*, ICFP 2013, pp. 36–51, New York, NY, USA, 2013. ACM.

[3] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. In *Journal of Functional Programming*, Volume 19 Issue 5, pp. 509–543, New York, NY, USA, September 2009.

[4] Kenichi Suzuki, Oleg Kiselyov, and Yukiyoshi Kameyama. Finally, safely-extensible and efficient language-integrated query. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, pp. 37–48, New York, NY, USA, 2016. ACM.

[5] Simon Jones, Peyton and Philip Wadler. Comprehensive comprehensions comprehensions with 'order by' and 'group by'. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, Haskell '07, pp. 61–72, New York, NY, USA, 2007. ACM.

[6] Ezra Cooper. The script-writer's dream: How to write great sql in your own language,and be sure it will succeed. In *Proceedings of the 12th International Symposium on Database Programming Languages*, DBPL '09, pp. 36–51, Berlin, Heidelberg, 2009. Springer-Verla.

[7] Tatsuya Katsusima and Oleg Kiselyov. Sound and efficient language-integrated query. In *Asian Symposium on Programming Languages and Systems*, APLAS 2017, pp. 364–383, LNCS 10695, 2017. Springer.

[8] James Cheney, Sam Lindley, and Philip Wadler. Query shredding: Efficient relational evaluation of queries over nested multisets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pp. 1027–1038, New York, NY, USA, 2014. ACM.

[9] Microsoft corporation. the linq project: .net language integrated query. White paper, September 2005.

付録 A

正規化規則

付録では、Suzuki らが対象言語を QueΛ として Cheney らの正規化規則を記述した正規化規則に加え、本研究で追加した正規化規則を図 A.1、図 A.2 に示す。なお、ネストされた構造のクエリをフラットな構造のクエリへ変換する規則を **Stage 1** として記述し、$M \rightsquigarrow N$ と表記する。また、SQL に直接変換できるように並べ替える規則を **Stage 2** として記述し、$M \hookrightarrow N$ と表記する。

Stage 1

$$
\begin{array}{rcll}
(\lambda x.N)\ M & \rightsquigarrow & N[x := M] & \text{(ABS-}\beta\text{)} \\
\langle \overrightarrow{l\ =\ M} \rangle.l_i & \rightsquigarrow & M_i & \text{(RECORD-}\beta\text{)} \\
\mathsf{for}\ (x \leftarrow \mathsf{yield}\ M)\ N & \rightsquigarrow & N[x := M] & \text{(FORYIELD)} \\
\mathsf{for}\ (x \leftarrow \mathsf{for}\ (y \leftarrow L)\ M)\ N & \rightsquigarrow & & \\
\multicolumn{3}{r}{\mathsf{for}\ (y \leftarrow L)\ \mathsf{for}\ (x \leftarrow M)\ N \quad (\text{if } y \notin FV(N))} & \text{(FORFOR)} \\
\mathsf{for}\ (x \leftarrow \mathsf{where}\ L\ M)\ N & \rightsquigarrow & \mathsf{where}\ L\ (\mathsf{for}\ (x \leftarrow M)\ N) & \text{(FORWHERE)} \\
\mathsf{for}\ (x \leftarrow [\,])\ N & \rightsquigarrow & [\,] & \text{(FOREMPTY}_1\text{)} \\
\mathsf{for}\ (x \leftarrow M_1 \uplus M_2)\ N & \rightsquigarrow & & \\
\multicolumn{3}{r}{\mathsf{for}\ (x \leftarrow M_1)\ N \uplus \mathsf{for}\ (x \leftarrow M_2)\ N} & \text{(FORUNIONALL}_1\text{)} \\
\mathsf{where\ true}\ M & \rightsquigarrow & M & \text{(WHERETRUE)} \\
\mathsf{where\ false}\ M & \rightsquigarrow & [\,] & \text{(WHEREFALSE)} \\
\mathsf{gfor}\ (z \leftarrow \overrightarrow{k}\ (y \leftarrow \mathsf{for}\ (x \leftarrow M)\ N)\ L)\ O & \rightsquigarrow & & \\
\multicolumn{3}{r}{\mathsf{gfor}\ (z \leftarrow \overrightarrow{k}\ ((y \leftarrow N), (x \leftarrow M))\ L)\ O} & \text{(GFORFOR)} \\
\mathsf{gfor}\ (z \leftarrow \overrightarrow{k}\ (y \leftarrow \mathsf{where}\ M\ N)\ L)\ O & \rightsquigarrow & & \\
\multicolumn{3}{r}{\mathsf{gfor}\ (z \leftarrow \overrightarrow{k}\ (y \leftarrow N)\ (L \wedge M))\ O} & \text{(GFORWHERE)} \\
\mathsf{gfor}\ (z \leftarrow \overrightarrow{k}\ (y \leftarrow \mathsf{yield}\ M)\ L)\ N & \rightsquigarrow & N[y := M] & \text{(GFORYIELD)} \\
\mathsf{gfor}\ (z \leftarrow \overrightarrow{k}\ (y \leftarrow [\,])\ L)\ N & \rightsquigarrow & [\,] & \text{(GFOREMPTY}_1\text{)} \\
\mathsf{having\ true}\ M & \rightsquigarrow & M & \text{(HAVINGTRUE)} \\
\mathsf{having\ false}\ M & \rightsquigarrow & [\,] & \text{(HAVINGFALSE)}
\end{array}
$$

図 A.1　正規化規則 (Stage 1)

Stage 2

$$\mathsf{for}\ (x \leftarrow M)\ (N_1 \uplus N_2) \hookrightarrow \mathsf{for}\ (x \leftarrow M)\ N_1 \uplus \mathsf{for}\ (x \leftarrow M)\ N_2 \quad (\text{FORUNIONALL}_2)$$

$$\mathsf{for}\ (x \leftarrow M)\ [\,] \hookrightarrow [\,] \quad (\text{FOREMPTY}_2)$$

$$\mathsf{where}\ L\ (M \uplus N) \hookrightarrow (\mathsf{where}\ L\ M) \uplus (\mathsf{where}\ L\ N) \quad (\text{WHEREUNION})$$

$$\mathsf{where}\ L\ [\,] \hookrightarrow [\,] \quad (\text{WHEREEMPTY})$$

$$\mathsf{where}\ L\ (\mathsf{where}\ M\ N) \hookrightarrow \mathsf{where}\ (L \wedge M)\ N \quad (\text{WHEREWHERE})$$

$$\mathsf{where}\ L\ (\mathsf{for}\ (x \leftarrow M)\ N) \hookrightarrow \mathsf{for}\ (x \leftarrow M)\ (\mathsf{where}\ L\ N) \quad (\text{WHEREFOR})$$

$$\mathsf{for}\ (x \leftarrow M)\ \mathsf{gfor}\ (z \leftarrow \overrightarrow{k}\ (\overrightarrow{y \leftarrow N})\ L)\ O \hookrightarrow \mathsf{gfor}\ (z \leftarrow (\overrightarrow{k}, x_1)\ ((\overrightarrow{y \leftarrow N}), (x \leftarrow M))\ L)\ O \quad (\text{FORGFOR})$$

$$\mathsf{where}\ L\ \mathsf{gfor}\ (z \leftarrow \overrightarrow{k}\ (\overrightarrow{y \leftarrow N})\ M)\ O \hookrightarrow \mathsf{gfor}\ (z \leftarrow \overrightarrow{k}\ (\overrightarrow{y \leftarrow N})\ M)\ \mathsf{having}\ L\ O \quad (\text{WHEREGFOR})$$

$$\mathsf{gfor}\ (z \leftarrow \overrightarrow{k}\ (\overrightarrow{y \leftarrow M})\ L)\ [\,] \hookrightarrow [\,] \quad (\text{GFOREMPTY}_2)$$

$$\mathsf{having}\ L\ [\,] \hookrightarrow [\,] \quad (\text{HAVINGEMPTY})$$

$$\mathsf{having}\ L\ (\mathsf{having}\ M\ N) \hookrightarrow \mathsf{having}\ (L \wedge M)\ N \quad (\text{HAVINGHAVING})$$

図 A.2　正規化規則 (Stage 2)